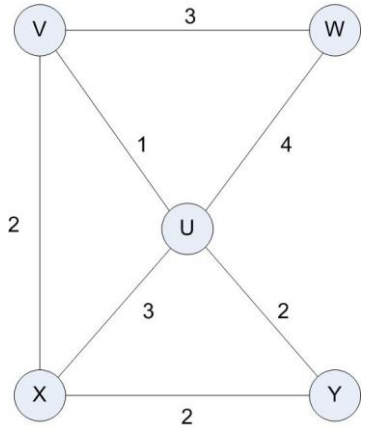
Algorithm: Lab13 (By Sujiv Shrestha ID:610145)
Problem 1.

1. Carry out the steps of Dijkstra's algorithm to compute the length of the shortest path between vertex V and vertex Y in the graph below. Your final answer should consist of three elements:
   a) The length of the shortest path from V to Y
   b) The list A[] which shows shortest distances between V and every other vertex
   c) The list B[] which shows shortest paths between V and every other vertex



Step1: X←{V}
A[V] ←0
B[V] ←{}

Step2: X={V}
Pool←{(V,W), (V,U), (V,X)}
Find minimum greedy length, min of the following
A[V]+wt(V,W) = 0+3 = 3
A[V]+wt(V,U) = 0+1 = 1
A[V]+wt(V,X) = 0+2 = 2

A[U]←1
X←{V,U}
B[U]←B[V]∪{(V,U)} = {(V, U)}

Step3: X={V, U}
Pool←{(V,W), (V,X), (U,X), (U,Y), (U,W)}
Find minimum greedy length, min of the following
A[V]+wt(V,W) = 0+3 = 3
A[V]+wt(V,X) = 0+2 = 2
A[U]+wt(U,X) = 1+3 = 4
A[U]+wt(U,Y) = 1+2 = 3
A[U]+wt(U,W) = 1+4 = 5

A[X]←2
X←{V,U,X}
B[X] ←B[V]∪{(V,X)} = {(V, X)}

Step4: X={V, U, X}
Pool←{(V,W), (U,Y), (U,W), (X,Y)}
Find minimum greedy length, min of the following
A[V]+wt(V,W) = 0+3 = 3
A[U]+wt(U,Y) = 1+2 = 3

A[U]+wt(U,W) = 1+4 = 5
A[X]+wt(X,Y) = 2+2 = 4

A[W]←3
X←{V,U,X,W}
B[W] ←B[V]∪{(V,W)} = {(V, W)}

Step5: X={V, U, X, W}
Pool←{(U,Y), (X,Y)}
Find minimum greedy length, min of the following
A[U]+wt(U,Y) = 1+2 = 3
A[X]+wt(X,Y) = 2+2 = 4

A[Y]←3
X←{V,U,X,W,Y}
B[Y]←B[U]∪{(U,Y)} = {(V,U), (V, Y)}

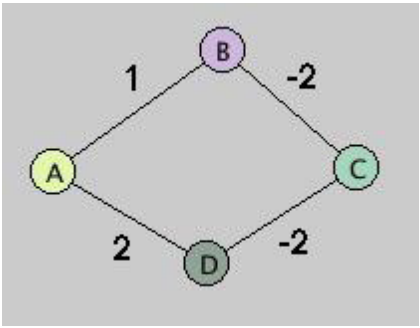
Solution:
A[V]=0, A[U]=1, A[X]=2, A[W]=3, A[Y]=3
B[V]={}, B[U] = {(V, U)}, B[X] ={(V,X)}, B[W]={(V,W)}, B[Y]={(V,U),(V,Y)}

Shortest length between V and Y is = V-U-Y (distance = 3)

Problem 2.

2. Points about Dijkstra's Algorithm

a. What is the shortest path from A to C in the graph below?

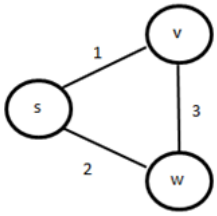


(Dijkstra's Algorithm does not deal with weighted graphs with negative weights.)
Answer: Apparently, the shortest path from A to C in the graph seems to be A-to-B-to-C as the sum of weights on edges along the path is -1 while its 0 for alternative path A-to-D-to-C. But, due to negative weight on edge in the undirected graph, another alternative path A-D-C-D-C will have total weight of edges on the path as -4 which is less than the previous. So, to eliminate this ambiguity, Dijkstra's algorithm has a requirement that none of the weights on edges should be negative.

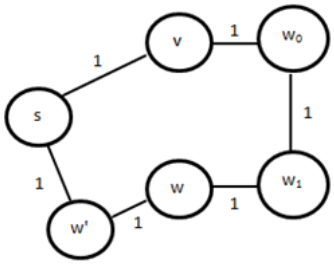
b.

Why is Dijkstra's approach to the shortest path problem better than simply using BFS, as described in the previous lesson?

[BFS approach: Making all edge weights = 1 is same as removing all weights. Perform BFS with start vertex s and compute distance to each vertex by returning its *level* in the BFS spanning tree. These computed values should be same as values found using Dijkstra]



↓BFS Style



Answer: Although the shortest path calculated from Dijkstra's algorithm and the BFS algorithm on modified equivalent graph results to same solution, Dijkstra's algorithm performs better simply because of the fact that the optimal running time for Dijkstra's algorithm is O(m*logn) while the running time for BFS is O(n+m) but for the BFS the 'n' (number of nodes) and 'm' (number of edges) increases according to the weights on the edges leaving it undecisive to determine the complexity class and eventually the running time grows with the weights. Also the modification of the graph to make the weights on each edge equal is an overhead task.

Problem3.

3. Describe an algorithm for deleting a key from a heap-based priority queue that runs in O(log n) time, where n is the number of nodes. (Hint: You may use auxiliary storage as the priority queue is built and maintained. Assume there are no two nodes have the same key.) This technique is needed for the optimized Dijkstra algorithm discussed in the slides.

Solution:

Deleting a key from a heap-based priority queue can be performed in following four steps:

1. Find the position of the key to be deleted.

2. Replace the key with the key of the last node.

3. Make the last node null

4. Restore the heap-order property (upheap or downheap)

The algorithm above works in similar way as removing the root node in priority-queue and runs in O(logn) except for the key search step that requires O(n). We can optimize this using a bucket

array for storing indices of keys so that searching the position of keys is performed in O(1) running time.

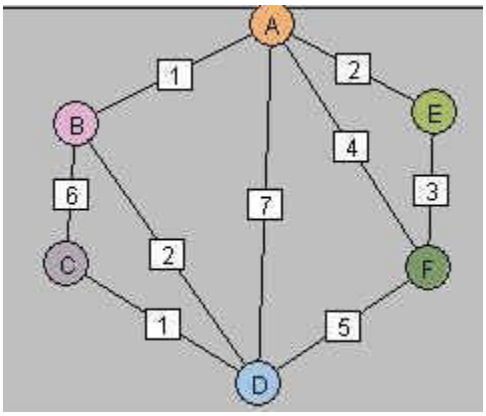
```
Algorithm: deleteKey(K)
    Input key K to be deleted from priority queue PQ
        (Assumption bucket Array I[ ] is maintained to store indices of keys in PQ)
    Output deletes the key K
    i←I[K]
    swap(i, n)              //swapping the key K with the key of last node
    I[K] ←null
    parent←PQ[i]
    child1←PQ[2*i+1]
    child2←PQ[2*i+2]
    if(i>0)then
        grandP←PQ[(i-1)/2]
    if(grandP>parent)then
        while (grandP> parent) do
            swap(i, (i-1)/2)
            i ← (i-1)/2
            parent←PQ[i]
            grandP←PQ[(i-1)/2]
            if(i=0) then
                return
    else
        while (parent>child1 or parent>child2) do
            if(child2>child1) then
                swap(i, 2*i+1)
                i ← 2*i+1
            else
                swap(i, 2*i+2)
                i ← 2*i+2
            parent←PQ[i]
            child1←PQ[2*i+1]
            child2←PQ[2*i+2]
            if(child1 = null and child2 = null) then
                return
    return

Algorithm: swap(i, j)
    Input indices i and j of keys to be swapped in PQ
    temp←PQ[i]
    PQ[i]←PQ[j]
    PQ[j]←temp

    temp←I[PQ[i]]
    I[PQ[i]]←I[PQ[j]]
    I[PQ[j]]←temp
    return
```

Problem4.

4. Carry out the steps of Kruskal's algorithm for the following weighted graph, using the tree-based DisjointSets data structure to represent clusters. Keep track of edges as they are added to T and show the state of representing trees through each iteration of the main while loop.



Step1: Sort the edges according to their weights

Sorted Edges: AB, CD, AE, BD, EF, AF, DF, BC, AD

T={}

Initialize cluster:

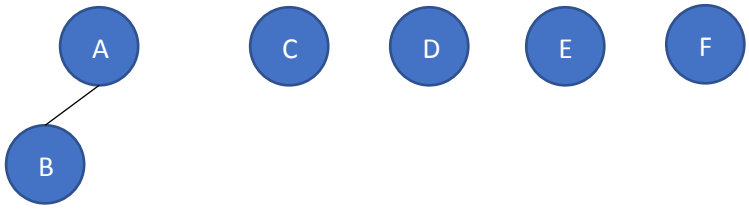
Step2: Sorted Edges: ~~AB~~, CD, AE, BD, EF, AF, DF, BC, AD

Check if C(A) ≠ C(B)

True, add AB to T and merge A and B

T={AB}

Cluster:



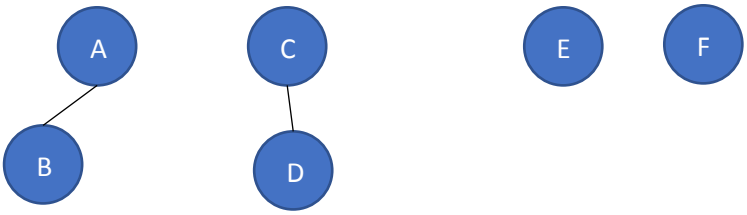
Step3: Sorted Edges: ~~AB~~, ~~CD~~, AE, BD, EF, AF, DF, BC, AD

Check if C(C) ≠ C(D)

True, add CD to T and merge C and D

T={AB,CD}

Cluster:



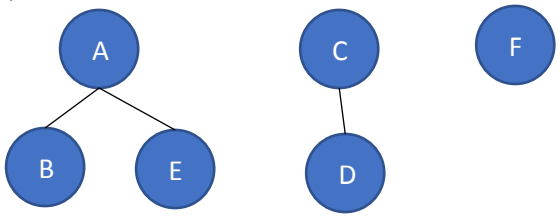
Step4: Sorted Edges: ~~AB~~, ~~CD~~, ~~AE~~, BD, EF, AF, DF, BC, AD

Check if C(A) ≠ C(E)

True, add AE to T and merge A and E

T={AB,CD,AE}

Cluster:



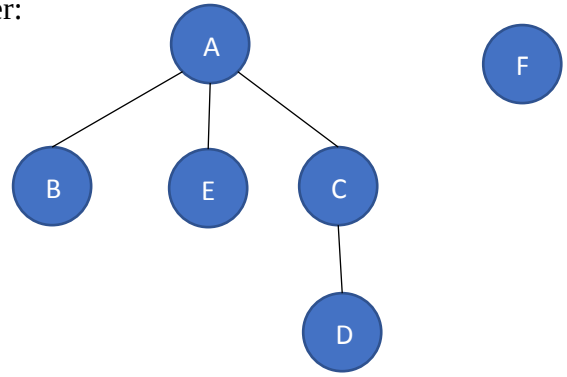
Step5: Sorted Edges: ~~AB~~, ~~CD~~, ~~AE~~, ~~BD~~, EF, AF, DF, BC, AD

Check if C(B) ≠ C(D)

True, add BD to T and merge A and C

T={AB,CD,AE,BD}

Cluster:



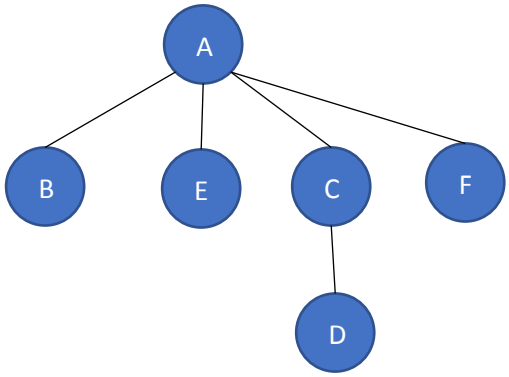
Step6: Sorted Edges: ~~AB~~, ~~CD~~, ~~AE~~, ~~BD~~, ~~EF~~, AF, DF, BC, AD

Check if C(E) ≠ C(F)

True, add EF to T and merge A and F

T={AB,CD,AE,BD,EF}

Cluster:



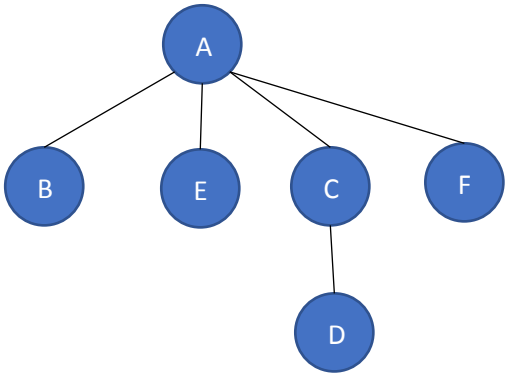
Step7: Sorted Edges: ~~AB~~, ~~CD~~, ~~AE~~, ~~BD~~, ~~EF~~, ~~AF~~, DF, BC, AD

Check if C(A) ≠ C(F)

False, ignore AF

T={AB,CD,AE,BD,EF}

Cluster:



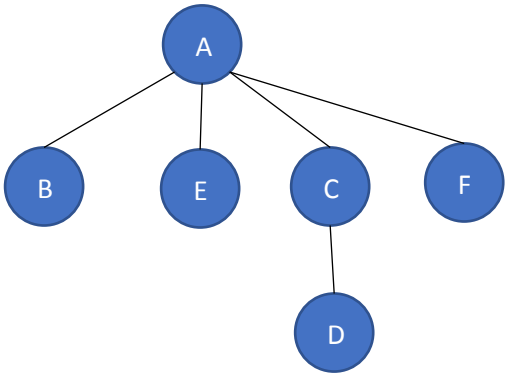
Step8: Sorted Edges: ~~AB~~, ~~CD~~, ~~AE~~, ~~BD~~, ~~EF~~, ~~AF~~, ~~DF~~, BC, AD

Check if C(D) ≠ C(F)

False, ignore DF

T={AB,CD,AE,BD,EF}

Cluster:



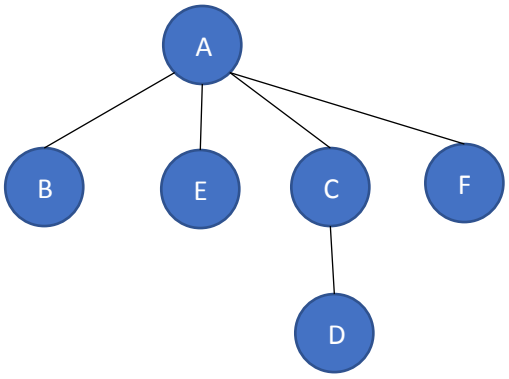
Step9: Sorted Edges: ~~AB~~, ~~CD~~, ~~AE~~, ~~BD~~, ~~EF~~, ~~AF~~, ~~DF~~, ~~BC~~, AD

Check if C(B) ≠ C(C)

False, ignore BC

T={AB,CD,AE,BD,EF}

Cluster:



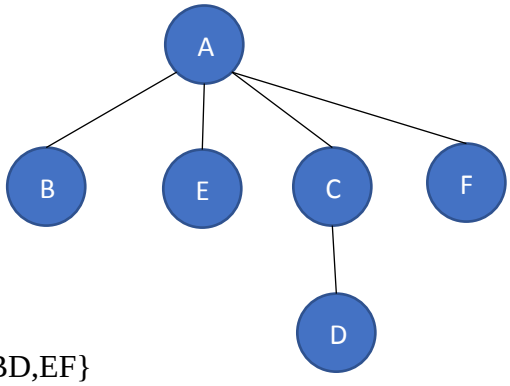
Step10: Sorted Edges: ~~AB~~, ~~CD~~, ~~AE~~, ~~BD~~, ~~EF~~, ~~AF~~, ~~DF~~, ~~BC~~, ~~AD~~

Check if C(A) ≠ C(D)

False, ignore AD

T={AB,CD,AE,BD,EF}

Cluster:



Minimum-spanning tree: T = {AB,CD,AE,BD,EF}